EXHIBIT 2

USENIX Association

Proceedings of the 5th Symposium on Operating Systems Design and Implementation

Boston, Massachusetts, USA
December 9–11, 2002

The Design and Implementation of Zap: A System for Migrating Computing Environments

Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh
Department of Computer Science
Columbia University
{sto8, dinesh, gongsu, nieh}@cs.columbia.edu

Abstract

We have created Zap, a novel system for transparent migration of legacy and networked applications. Zap provides a thin virtualization layer on top of the operating system that introduces pods, which are groups of processes that are provided a consistent, virtualized view of the system. This decouples processes in pods from dependencies to the host operating system and other processes on the system. By integrating Zap virtualization with a checkpoint-restart mechanism, Zap can migrate a pod of processes as a unit among machines running independent operating systems without leaving behind any residual state after migration. We have implemented a Zap prototype in Linux that supports transparent migration of unmodified applications without any kernel modifications. We demonstrate that our Linux Zap prototype can provide general-purpose process migration functionality with low overhead. Our experimental results for migrating pods used for running a standard user's X windows desktop computing environment and for running an Apache web server show that these kinds of pods can be migrated with subsecond checkpoint and restart latencies.

1 Introduction

Process migration is the ability to transfer a process from one machine to another. It is a useful facility in distributed computing environments, especially as computing devices become more pervasive and Internet access becomes more ubiquitous. The potential benefits of process migration, among others, are fault resilience by migrating processes off of faulty hosts, data access locality by migrating processes closer to the data, better system response time by migrating processes closer to users, dynamic load balancing by migrating processes to less loaded hosts, and improved service availability and administration by migrating processes before host maintenance so that applications can continue to run with minimal downtime.

Although process migration provides substantial potential benefits and many approaches have been considered [24], achieving process migration functionality has been difficult in practice. Toward this end, there are four important goals that need to be met. First, given the large number of widely used legacy applications, applications should be able to migrate and continue to operate correctly without modification, without requiring that they be written using uncommon languages or toolkits, and without restricting their use of common operating system services. For example, networked applications should be able to maintain their network connections even after being migrated. Second, migration should leverage the large existing installed base of commodity operating systems. It should not necessitate use of new operating systems or substantial modifications to existing ones. Third, migration should maintain the independence of independent machines. It should avoid creating residual dependencies that limit the utility of process migration by requiring machines where a process was previously executed to continue to service a process even after it has migrated to another machine. Fourth, migration should be fast and efficient. Overhead should be small for normal execution and migration.

To overcome limitations in previous approaches to general-purpose process migration, we have created Zap. Zap provides a thin virtualization layer on top of the operating system that introduces a *PrOcess Domain* (pod) abstraction. A pod provides a group of processes with a private namespace that presents the process group with the same virtualized view of the system. This virtualized view associates virtual identifiers with operating system resources such as process identifiers and network addresses. This decouples processes in a pod from dependencies on the host operating system and from other processes in the system.

Zap virtualization is integrated with a checkpoint-restart mechanism that enables processes within a pod to be migrated as a unit to another machine. Since pods are independent and self-contained they can be migrated freely without leaving behind any residual state after migration, even when migrating network applications while preserving their network connections. Zap can therefore allow applications to continue executing after migration even if the machine on which they previously executed is no longer available. In using a checkpoint-restart approach, Zap not only supports process migration, but also allows processes to be suspended to secondary storage and transparently resumed at a later time.

Going beyond simple migration, this functionality can be useful in many ways, including fast creation of user sessions and simpler, more dynamic system configuration.

Zap is designed to support migration of unmodified legacy applications while minimizing changes to existing operating systems. This is done by leveraging loadable kernel module functionality in commodity operating systems that allows Zap to intercept system calls as needed for virtualization and save and restore kernel state as needed for migration. Zap's compatibility with existing applications and operating systems makes it simple to deploy and use. We have implemented a Zap prototype as a loadable kernel module in Linux that supports transparent migration, without any kernel modifications, among separate machines running independent Linux operating systems; it does not require a single-system image across all machines. Our experimental results on our Linux Zap prototype demonstrate that it can provide general-purpose process migration functionality with low overhead.

This paper focuses on the design and implementation of the Zap virtualization and migration mechanisms. Section 2 describes related work. Section 3 describes the pod abstraction provided by Zap. Section 4 describes the architecture of Zap and the mechanisms that support the pod abstraction. Section 5 presents an overview of our implementation of Zap in Linux. Section 6 presents experimental results evaluate the overhead associated with Zap virtualization and migration mechanisms and demonstrate the utility of Zap for migrating legacy and network applications. Finally, we present concluding remarks and directions for future work.

2 Related Work

Many research operating systems have been developed that implemented process migration mechanisms, with a focus on using migration for load balancing. These systems include Accent [31], Amoeba [25], Charlotte [6], Chorus [33], MOSIX [7], Sprite [12], and V [11]. These operating systems provided a single system image across a cluster of machines and providing migration throughout the cluster through careful kernel design to provide a global namespace and location-transparent execution. Process state such as IPC, open files, and system calls in some cases are typically handled by forwarding requests to a home node on which the process originated. If the home node fails, migrated processes running on other nodes may fail as well. Although providing a single cluster operating system can simplify system management, these kinds of systems require new operating systems or substantial changes to existing ones, which have limited their deployment. Furthermore, these approaches do not work in the context of increasingly common clusters of independent machines, each with its own operating system.

Several systems have been developed to support process migration at the user-level and can be run on unmodified commercial operating systems. These systems include Condor [22], CoCheck [29], libckpt [28], and MPVM [10]. These systems are primarily intended for executing long-running applications on a cluster of machines. However, because there is no kernel support for process migration, these systems require processes to be well-behaved in order to migrate, which means that such processes cannot use common operating system services such as inter-process communication. This severely limits the kind of applications that can be used with such systems.

Several systems have been developed that provide migration using object-based approaches. These systems include Abacus [5], Emerald [19], Globus [13], Legion [14], and Rover [18]. These systems are designed as programming languages or middleware toolkits that typically require explicit programmer control to utilize migration. By operating at a higher-level of abstraction, these systems can reduce the amount of state that needs to be recorded and moved to migrate an application. However, these systems require applications to be rewritten using new programming language environments. As a result, they cannot migrate legacy applications.

Virtualization at the operating system level has been proposed as a mechanism for supporting process migration. Zap virtualization was inspired in part by capsules [36], an abstraction that provided a private namespace to a group of processes that can be migrated as a unit. However, capsules did not support migration of networked applications while preserving their open network connections. Unlike Zap, implementing capsules required extensive operating system changes. Whereas the capsule approach restructured the operating system to achieve its goals, Zap seeks to be compatible with existing operating systems by virtualizing the operating system interface while minimizing changes to the operating system. Operating system virtualization has also been explored in vOS [9] to provide process migration for simple, non-networked Windows applications.

Virtual machine monitors (VMMs) can also be used as a mechanism for process migration [21, 35]. Virtual machine monitors such as VMware [1] virtualize at the hardware level to encapsulate an entire operating system environment such that it can be suspended and resumed. The operating system environment can be migrated from one machine to another assuming sufficient similarities in those system architectures. By leveraging VMware, even Microsoft Windows applications can be migrated without operating system or application changes, though the problem of migrating networked applications with open connections has not yet been addressed. Capsules have been recently applied in this context [35]. Because

VMMs operate below the operating system, they cannot take advantage of mechanisms specific to a given operating system to reduce the cost of migration and are limited to migrating an entire machine as opposed to a few processes. Furthermore, all applications to be migrated must be run using a VMM. Section 6 shows that the resulting cost of using VMMs can be substantially higher migration and runtime overhead.

Previous approaches to process migration do not effectively support networked applications. However, a variety of other approaches have been proposed to provide mobility for network communications. These approaches can be loosely categorized as network layer solutions [8, 17, 26], transport layer solutions [38], proxy-based solutions [23], and socket library wrapper solutions [30, 41]. Network layer solutions do not provide mobility for individual end-to-end transport connections; they only allow mobility at the level of an entire host. The proposed transport layer solution requires changes in the transport protocol and therefore is difficult to deploy. Existing proxy-based solutions are usually tied to a specific transport protocol (e.g., TCP) and their "connection switching and splicing" function incurs high overhead. Socket library wrapper solutions duplicate many transport protocol functions that result in high overhead.

3 The Pod Abstraction

The goal of Zap is to support migratable computing environments in the context of today's networked computing infrastructure. In this infrastructure, network file servers are typically used to store applications and user data. These servers are then accessible to personal computers or compute servers, where application processing takes place. A key characteristic of this computing environment is that the compute machines typically run completely independently of one another, each running its own independent operating system. Zap seeks to enable users to continue to use such a computing infrastructure as they normally do, but with the added feature of being able to have their computing sessions migrate across machines.

To support transparent process migration from one independent machine to another, Zap must address three key requirements regarding resource consistency, resource conflicts, and resource dependencies. The first requirement is the need to preserve resource naming consistency. An operating system contains numerous identifiers for its resources, including process IDs (PIDs), file names, and socket ports. Since neither the operating system nor the typical application were designed to support process migration, they both assume that these identifiers will remain constant throughout the life of the process. It is not unusual for a process to take note of an identifier given to it and store it in memory or on a file. In migrating a process from one machine to another, it is important to maintain consistent names for these resource identifiers to ensure that the process continues to function correctly.

The second requirement is the need to avoid potential resource naming conflicts in the presence of migrating processes. Since operating system resource identifiers are unique only at the system level, it is a trivial task for an operating system to produce unique identifiers merely by examining its current state and picking a candidate identifier that is not in use. The problem that can arise when a process is migrated to a new host system is that its process identifier could already be in use at the new host system. For example, a conflict will arise if a process with PID 20 is migrated into a system that already contains a process with PID 20. Besides aborting the migration, there are only two approaches at this point: (1) violate resource naming consistency and change the PID, which can cause common applications which have already stored their previous PIDs to fail because PID value has changed unexpectedly, or (2) wait and try again later after the PID resource becomes free, which can result in the migration never being able to occur since there is no way to know how long it will be before the resource becomes free.

The third requirement is the need to avoid creating dependencies among components of the system that cannot be easily severed when a process is migrated. For example, a process could attempt to share an area of memory with an operating system component such as another process, making it impossible to migrate the particular process unless: (1) the other component sharing memory is also migrated simultaneously, or (2) a proxy is left behind on the original host so that any updates of this particular shared memory area will be relayed through a network connection. The first approach may require larger and larger portions of the operating system to migrate at once. As the number of cross dependencies grows, the number of independently migratable processes decreases. The second approach is also unfavorable because application developers assume that accessing a shared memory area will be fast compared to network speeds. In addition, reliance on the original host machine is never removed, so that some of the potential advantages of process migration, such as freeing up a machine for maintenance, are lost.

To address these three requirements, Zap introduces a *pod* (PrOcess Domain) abstraction, which provides a collection of processes with a host-independent virtualized view of the operating system. Pods are self-contained units that can be suspended to secondary storage, migrated to another machine, and transparently resumed. A pod can contain any number of processes. For example, a pod can encapsulate all of the processes corre-

sponding to a user's computing session. The abstraction provides the same application interface as the underlying operating system so that legacy applications can execute in the context of a pod without any modification. Processes within a pod can make use of all available operating system services, just like processes executing in a traditional operating system environment.

The main difference between a pod and a traditional operating system environment is that each pod has its own private, virtual namespace. The idea of a private, virtual namespace is surprisingly simple but has significant implications for supporting migratable computing environments. The pod namespace provides two key characteristics that facilitate the independence and mobility of pods.

First, the namespace provides consistent, virtual resource names in place of host-dependent resource names such as PIDs. Names within a pod are trivially assigned in a unique manner in the same way that traditional operating systems assign names, but such names are localized to the pod. Since the namespace is private to a given pod, there are no resource naming conflicts for processes in different pods. There is no need for the pod namespace to change when the pod is migrated, which allows pods to ensure that identifiers remain constant throughout the life of the process, as required by legacy applications that use such identifiers. Because pod namespaces are private, allowing a process to move from one pod to another could result in naming conflicts. As a result, processes are created inside of a pod and spend their entire lifetimes in the context of that pod; they are not allowed to leave one pod and join another.

Consider the following simple example of how the pod namespace aids in supporting process migration. The following excerpt of a C program is not atypical of a multithreaded application:

```
int iChildPID;

if (iChildPID=fork()) {
  /* parent does some work */
  waitpid(iChildPID);/* Wait for the child
                        process to exit */
} else {
  /* child does some work */
  exit(0);
}
```

Migrating this program from a system A to a system B is trivial in the context of pods. If the child had a virtual PID of 172 in a pod on system A, it will still have the same PID in the same pod after it is migrated to system B. Since the pod namespace is private, there is no possible naming conflict. However, migrating even this simple program without pods could be problematic. Without pods, the PID namespace is global on a system. If system B already had a process running with PID 172, the child process in the above program would need to be assigned a new PID when it moves from system A. The parent process, however, expects the child process to have a PID of 172. When it calls `waitpid(iChildPID)`, it will be waiting for the wrong process.

Second, the private namespace masks out resources that are not contained within the pod, including processes outside of the pod. Pod namespace masking creates independence among elements that can migrate to avoid creating ties between components of the system that cannot be easily severed when an environment is migrated. A pod logically groups processes running on an operating system into three classes: processes inside the pod, special system processes outside the pod, and other processes outside the pod. Processes inside a pod appear to one another as normal processes that can communicate using traditional IPC mechanisms. Special system processes outside the pod that are consistently named across different machines, such as the init process in Linux, are viewable from within a pod but cannot interact with processes within the pod using IPC. In particular, the init process, or its equivalent system process that serves as the parent of orphaned children, is made viewable within a pod to serve as the parent of processes within a pod that have no parent process within the pod. Other processes outside a pod do not appear in the namespace and are therefore not able to interact with processes inside a pod using IPC mechanisms such as shared memory and signals. Instead, processes outside the pod can only interact with processes inside the pod using network communication and shared files that are normally used to support process communication across machines. Unlike IPC mechanisms, these communications mechanisms can avoid imposing host-specific dependencies that would limit the ability to migrate a pod.

4 Zap Architecture

Zap provides an architecture that supports the pod abstraction in the context of commodity operating systems. This is difficult to achieve because there are important mismatches and conflicts between the pod abstraction needed for process migration and current operating system design. For example, PIDs are determined by the host operating system which has no knowledge of pods, whereas pods require their own private PIDs for their respective processes. Processes running on the same operating system are free to communicate using IPC, whereas pods disallow such communication across pods to avoid creating host dependencies.

Zap provides pod functionality using commodity operating systems by inserting a thin virtualization layer between applications and the operating system. This virtualization layer is used to translate between pod namespaces and the underlying host operating system

namespace, and protect processes within a pod from dependencies on processes outside the pod. We refer to pod resource names as *virtual names* and operating system resource names as *physical names*. Each pod virtual name corresponding to an operating system resource is mapped to an underlying operating system physical name. For example, a process is given a pod virtual PID that must be mapped to the physical PID used by the operating system. Zap virtualization is done by intercepting system calls and translating their arguments and return values as needed. When pod virtual names such as PIDs are passed to the operating system via system calls, Zap first translates those names to the corresponding operating system resource names then passes those physical resource names to the operating system. Similarly, when operating system resource names are returned via system calls to processes in a pod, Zap first translates those names to the corresponding pod resource names then passes those pod virtual names up to applications. Operating system resources without corresponding pod virtual names are masked out of the respective pod's namespace. Although Zap virtualization provides the flexibility of using virtual names that are distinct from the corresponding physical names, it does not preclude using virtual names that are the same as physical names. For instance, the virtual and physical PID for the init process are the same since the physical PID for init is always 1.

Zap virtualization is coupled with a checkpoint-restart mechanism to suspend, migrate, and resume pods and their associated processes. Zap uses a checkpoint-restart approach for the following four reasons. First, it is simpler to implement than other migration mechanisms such as demand paging. Second, it avoids leaving behind residual components after migration since all of the required state is in a checkpointed image that is simply moved for migration. Third, it enables flexible mechanisms for migrating the pod state, whether it be by sending it over a network or storing it on a disk that is moved from one place to another. Fourth, it enables pods to be checkpointed, suspended to secondary storage, and stored for future use.

To migrate a pod, Zap first suspends the pod by stopping all processes in the pod, saving the virtualization mappings, and saving all process state, including memory, CPU registers, open file handles, etc. The saved pod state can be digitally signed to avoid tampering and can then be moved to a new host in any number of ways. Zap defines an abstract I/O interface that is used for checkpointing and restarting a pod. The I/O interface has been defined specifically to allow only sequential writing (for checkpoint) and reading (for restart) of the image data. This ensures that checkpoint-restart can occur through the largest set of mediums, including ones which may not necessarily support two-way communication or which have extremely slow random-access speed. This allows data to be streamed to and from a tape or a network socket during migration, enabling pipelining of checkpoint-restart operations. On the new host, Zap resumes the pod by first restoring the pod virtualized environment, then restoring processes in a stopped state. Zap must then create necessary virtualization mappings pertaining to the stopped processes, such as remapping the virtual PID of a process to its new physical PID on the host. Finally, Zap enables the processes to continue executing in the restored pod environment.

The Zap architecture consistently applies the principles of pod namespaces, virtualization, and migration to a complete set of operating system resources. Table 1 provides an overview of how pod principles are applied to different resources. Section 4.1 discusses process identifiers and IPC resources. Section 4.2 considers memory resources. Section 4.3 focuses on file system resources. Section 4.4 introduces device resources. Section 4.5 overviews network resources. Section 4.6 describes pod administration and how pods are created and used.

Resource	Virtual Names	Migration State
Process State and IPC	Process and group IDs, IPC keys and IPC IDs	Process and group IDs, CPU registers, signal handlers, IPC keys and state, pipes and written, but not yet read by peer data in pipes
Memory	Memory addresses (already supported by OS)	Memory mappings and contents of data pages
File System	Directory structure, per-pod private directory, per-pod `/proc`	Opened files which have been unlinked, directories and files not mounted on a network file system
Devices	Per-pod `/dev`	Device names and state (device specific)
Network	Per-pod IP address, virtual network connections	Socket state, remote endpoint and virtual address pairs, queued but unsent data at the transport layer

Table 1: Pod principles applied to resources

4.1 Process IDs and IPC Virtualization and Migration

For each pod, Zap provides unique namespaces for process resources, including PIDs and keys for IPC mechanisms such as semaphores, shared memory, and message queues. Values from within these spaces are assigned the same way that they are by the operating system and are maintained consistently across migration.

Zap virtualization employs two types of hash tables that can be quickly indexed to translate these resource

names between private pod namespaces and the operating system namespace. One is a system-wide hash table indexed by physical identifiers on the host operating system that returns the corresponding pod and virtual identifier. The other is a per-pod hash table indexed by virtual identifiers specific to a pod that returns the corresponding physical identifiers. When a system call is made that uses one of the identifiers in question, Zap replaces all the parameters that refer to virtual identifiers for the current pod with physical identifiers. Zap then invokes the system call, and upon return from the call replaces all physical identifiers returned by the system with virtual ones by looking them up in the system-wide physical-to-virtual hash table.

Consider PID virtualization as an example. Zap provides a host system-wide process identifier hash table indexed by the physical process PID and a process identifier hash table for each pod indexed by the virtual process PID provided by the respective pod. System calls that manipulate PIDs are intercepted, including `getpid`, `getgid`, `fork`, `kill`, etc. In system calls like `getpid`, the returned physical PID is translated into a virtual one and in system calls like `kill`, the virtual PID argument is translated into the physical one before passing it on to the kernel for processing.

Since Zap intercepts system calls that would manipulate process resource identifiers, such as the call to attach to a particular area of shared memory or to send a signal to a process, Zap can trivially limit the successful calls to those that use valid identifiers within their context. Within a pod, a valid identifier is a virtual identifier that was created within the pod. Outside of the pod environment, the system call will reject any requests to physical resources which belong to pods. This disallows a process on the host operating system from creating dependencies between pods and the host operating system.

To migrate a pod and its processes, Zap checkpoints data that pertains to process resources which is then used to restart the pod after migration. The data that is saved includes Zap virtualization state contained in the hash tables, process state for processes in the pod, and state associated with IPC mechanisms used by processes in the pod. Process state saved includes CPU registers, process credentials, process signal handlers and any pending signals, and process, group and session identifiers as well as the process's controlling terminal.

IPC mechanism state saved includes data associated with mechanisms used to access external entities, such as TCP/IP sockets, as well as IPC mechanisms used to communicate with other processes within the pod, such as Unix sockets, semaphores, shared memory areas, and unnamed pipes. Shared memory migration is only slightly different from typical memory migration discussed in Section 4.2. They are similar in that shared memory contains an address range as well as contents to fit within that range, however, shared memory requires a little more attention. First, a shared memory region, like many of the other IPC mechanisms is identified by a key that allows disparate processes to connect to the same resource. In addition, if multiple processes refer to the same shared memory region, they refer to the same contents as well. It is sufficient to save and restore the contents for the first process in the pod that uses the memory area, then for all other processes and mappings of that area, only a reference is needed.

When a pod is migrated and restarted on a target machine, Zap recreates the pod virtual environment and its processes. Zap creates new processes on the machine into which it restores the checkpointed process data. Zap keeps track of the new operating system identifiers corresponding to the new resources created and updates its system-wide and per-pod hash tables with this information to reconstruct the Zap virtualization state in the context of new operating system resources used by the pod on the target machine. Zap ensures that the pod environment is updated before enabling the processes in the pod to start executing again.

4.2 Memory Virtualization and Migration

Because memory state associated with processes is in general implicitly virtualized by the operating system, Zap only needs to provide virtualization support for mechanisms which allow a particular area of memory to be shared by multiple processes. Specifically, a memory location can be mapped to a file, shared as a result of process creation, and shared through IPC shared memory. File-mapped memory is handled implicitly by Zap's file namespaces, discussed in Section 4.3. Memory shared due to process creation is handled implicitly by automatically including child processes in the same pod as the parent so that they both share the same namespace. IPC shared memory is handled by pod virtualization as previously discussed in Section 4.1.

To migrate a pod, Zap checkpoints the memory areas allocated by processes within the pod. As discussed further in Section 4.3, Zap assures that the same view of the file system is available to a pod on whatever machine the pod is executed. As a result, Zap need not checkpoint the contents code segments that belong to an executable. This includes the text pages of the process as well as those of dynamically linked libraries in use by the process. Migration of text pages is accomplished through saving references to the executable files as well as the virtual memory addresses to which they are mapped.

When a pod is restarted, the checkpointed memory data needs to be restored. For efficiency reasons, whenever possible, Zap maps the checkpointed data directly to memory, allowing the pages to be read in as they are

touched by the process. This can greatly improve the restart performance. If the checkpointed data is stored on a network file server, this can also reduce the network utilization of a process that will not necessarily touch all of its pages after restart.

4.3 File System Virtualization and Migration

To support migration of processes within pods, Zap must provide each pod with a consistent, location-independent view of the file system that is available on all hosts. One way this could be done would be to associate each pod with its own complete file system, which is migrated whenever the pod is migrated. Given that file systems can be many gigabytes, this would result in a substantial amount of state having to be migrated each time a pod moves from one machine to another making the approach impractical. Another way this could be done would be to have a global file system across all machines where a pod could be located. This removes the need to copy files from one machine to another since all files would be network accessible. However, ensuring a consistent global root file system across all machines is impractical as most machines are not configured this way in practice.

Zap takes a different approach to providing each pod with a consistent, location-independent view of the file system. Zap provides each pod with its own virtualized file system and corresponding private file system namespace. Zap then leverages widely used distributed file systems such as NFS [2] to store file data and mount such systems within the pod's virtual file system hierarchy. More specifically, when a pod is created or moved to a host, a private directory named according to a pod identifier is created on the host to serve as a staging area for the pod's virtual file system. Zap ensures that this directory is not accessible by processes on the host machine that are not in the given pod. Within this directory, the various network-accessible directories that the pod is configured to access will be mounted from a network file server. Minimally from a Unix-centric viewpoint, this set of directories would include `/etc`, `/lib`, `/bin`, and `/usr`. Each pod must also be configured with a `/tmp` that is private to the pod and is discussed further below. Zap then uses the `chroot` call to set the staging area as the root directory for the pod, thereby achieving file system virtualization with modest performance overhead. Zap virtualizes `chroot` to prevent processes within a pod from breaking out of their virtual file system environment. This approach takes advantage of distributed file systems to reduce file state that would need to be moved during migration without requiring a global file system across all host machines.

Zap can use network file servers to support many pods running on many machines at the same time. This is simple to do for files such as common executables which are used in the same way by processes in all pods. In some cases though, it is desirable and necessary to have files and directories that are specific to a given pod. For this purpose, Zap introduces private pod directories. A private pod directory is created when a pod is initially created and destroyed once the pod is finally destroyed. The private directory is only used by the given pod. The directory can be created on the network file server then mounted within the pod virtual file system as `/private-pod`. When a private pod directory is created, it may optionally be pre-populated with data from a template directory.

Private pod directories can be useful for supporting directories such as `/tmp` or `/local` that are typically local to a machine. It is important that such directories are private to avoid naming conflicts that would otherwise arise in the file system due to the way some legacy applications name files. For example, some server applications store their PIDs in the `/tmp` directory in a file with the server name as the filename and `.pid` as the file extension. If `/tmp` is shared by two pods that happen to both have instances of such a server running, a filename conflict will result. These conflicts arise due to processes in separate pod namespaces sharing the same file namespace and occur with directories such as `/tmp` that are typically local to a machine. Zap avoids this problem by storing `/tmp` and `/local` in the private pod directory when a pod and its virtual file system are created. Whereas `/tmp` is initially empty, `/local` can be populated by transferring over files from a template directory on the file server. Note that Zap does not have to use a network file server for such these private directories but could instead store them locally on the host machine as a subdirectory of the pod's virtual file system. However, this would require that the files be migrated as well when the pod is migrated.

Private pod directories can also be useful for allowing per-pod application configurations without having to duplicate the application file hierarchy. When some files or subdirectories used by a common application need to be specific to a given pod, these files can be easily configured as symbolic links to files in the respective private pod directories. For example, to install a web server that is available to all pods, an administrator could install the web server in a global `/usr/local/apache` directory, and make the conf directory within it a symbolic link to `/privatepod/apache/conf`. This will allow multiple pods to share one copy of the web server, which can be centrally managed and upgraded periodically to fix bugs and close up security holes, while each pod maintains its own configuration, allowing pods to point to log files and web pages anywhere on their file system.

In addition to network accessible files and private pod files, Zap must also consider special file systems such as the proc file system [27] in `/proc` and devices in `/dev`.

We briefly discuss /proc here and defer devices to Section 4.4. Each pod is given its own /proc by creating a special per-pod directory, specifically /proc/zap/pods/<pod_id>/proc, under the proc file system of the host machine and loopback mounting that area as /proc in the pod's virtual file system. The per-pod directory registers its own set of file operations with the proc file system for accessing files and directories instead of using the generic operations used when accessing files in /proc directly from the host. These pod /proc file functions are similar to the generic /proc functions except that they translate as needed to return system information in the context of the pod namespace. For example, the process directories in the pod /proc are listed by virtual pod PIDs instead of physical PIDs. All of the information in the Zap-specific area of /proc is created dynamically by combining pod virtualization information with system information from the operating system.

When a pod migrates, Zap flushes all cached data to disk and saves and restores the information it needs to reconstruct the pod virtual file system, including a list of all files opened by the processes within a pod and the access rights with which the files were opened. Zap in general does not need to save file contents because it leverages the use of a distributed file system to make the files available at the machine where the pod is resumed. Dynamically generated files in /proc also do not need to be saved since they can be recreated at the machine where the pod is resumed. In environments where a pod cannot access distributed file systems from all locations, Zap could easily be extended to package up the contents of the file system along with the checkpointed image of the pod.

Zap does checkpoint the contents of files that have been opened by a pod process and have been subsequently unlinked. This is because as soon as the pod processes are checkpointed, opened files are closed and the file system will free up the inodes associated with the files, losing the data that they contained. Unlinked files will no longer exist and will not be available when a pod is restarted. To address this problem, Zap checks the reference counters of inodes belonging to files opened by processes in a pod. If a pod has an unlinked file open, Zap saves the contents of the file and recreates the file when the pod is restarted, once again unlinking it after it has been opened.

4.4 Device Virtualization and Migration

Zap provides each pod with its own virtual /dev directory to provide a framework for supporting device virtualization and migration. Creating a unique /dev directory for a pod helps achieve two goals: first, it ensures that the pod cannot accidentally use any host-specific devices which may be difficult to migrate because they may be in an awkward interim state (such as the CD recording device while recording), or impossible to migrate because they are unavailable elsewhere (for example, just because one host is attached to an electron microscope, one cannot assume that all hosts will be attached to one). The second goal it achieves is naming resolution for certain files, for example, virtual tty names. When the system assigns a new tty to a process, it merely selects the next available tty number (for example, /dev/pts/2 or /dev/tty02). Without virtualizing the /dev namespaces, there would be no guarantee that the particular tty will be available for a given pod at its new location.

Each pod is given its own /dev by creating a special per-pod directory on the host machine and loopback mounting that area as /dev in the pod's virtual file system. Zap employs a device-specific plugin for each device, which registers a particular device within the virtual /dev directory and provides appropriate support for the given device. Device support needs to be addressed on a per-device basis; full migration support of devices is a difficult problem that is beyond the scope of this paper. Devices that are not explicitly supported by Zap are not included in a pod /dev directory, preventing processes within a pod from accessing them.

Zap defines three types of device support that could be provided as emulation, virtualization, and non-migratable. Emulation could be used to emulate a device in software. For example, a virtual console could be created and registered as /dev/console. When data is written to /dev/console, it could be redirected to a pre-defined text file accessible within the pod. Virtualization could be used to utilize an equivalent device on the host system from within the pod. For example, a virtual /dev/audio could be created. As the virtual /dev/audio is accessed, the virtual device driver would make note of any configuration changes requested of the audio device and pass them on to the host's audio device. When the pod is migrated to a new host, the audio device on the original host is closed and reset and the audio device on the new host is opened and configured with whatever state changes were invoked previously. Finally, non-migratable device driver could be created that passes all requests to the device on the host machine, but disallows migration so long as the device is in use. For example, when using a CD recording device, migrating in the middle of recording a CD would result in a CD with half its contents on being recorded on the original host and the other half on the new host. A non-migratable device driver could simply cause the pod migration to fail until the CD recorder device has been closed.

Device virtualization and migration greatly depend on the device and its capabilities. Currently, Zap explicitly supports pseudo-terminal devices, which allow one to log in remotely as well as have multiple terminals open

through such programs as `xterm`. Zap provides virtual ttys which require that, when a tty is opened, rather than the next available virtual tty for the host being returned, the next available virtual tty for the pod is returned. Upon migration, the migration process specifies the desired virtual tty number and the virtualization system will automatically map between the host's tty number and the pod's tty number.

4.5 Network Virtualization and Migration

Zap is designed to support migration of unmodified network applications running in pods using the existing network infrastructure without any modifications. Toward this end, Zap provides mechanisms to address three key issues: (1) enabling remote systems to locate and communicate with processes in the pod, (2) exposing application layer network interfaces to support persistent communication in the presence of migration, and (3) preserving consistent state at the transport layer to maintain an application's open connections persistently even after the application migrates.

To enable remote systems to locate and communicate with processes in the pod, Zap allows each pod to be assigned an external IP address that can be known to entities outside of a pod. The address is distinct from the IP address of the host machine where the pod is currently located. This external address changes as the given pod moves from host to host in the same way that a host machine's IP address changes when it changes network locations. The external IP address can be a routable one to enable a pod to provide network services such as a web server that can be accessed from an outside-initiated connection.

To allow a pod's network services to be accessed even if its external address changes due to migration, Zap leverages dynamic DNS [40] to maintain a name-to-IP relationship so that a pod's network services can be accessed by the same name even after migration. To avoid downtime due to migration, Zap can install a temporary proxy process just before a pod migrates, stall current and pending connections for a few seconds without negative effect, and then restart them once the pod is resumed on the new host machine. The number of incoming connections to the proxy will drop off as the TTL expires and when it gets below a defined threshold, the proxy terminates. The proxy only needs to run for a short time with a TTL of a few minutes, which does not adversely affect DNS caching performance [20].

To provide network interfaces for applications in a pod to support persistent communication in the presence of migration, Zap distinguishes between the external IP address and internal IP address perceived by a process running within the pod. Pods only allow applications within a pod to perceive the internal IP address directly. Pods can provide two types of network interfaces for determining the internal address seen by processes running within the pod: transient and persistent. Transient network interfaces assign the internal address equal to the current external IP address, which exposes the movement of a pod to applications within the pod. This is the default behavior, which works fine for most network applications because they do not require a fixed IP address for their connections to persist and function correctly even after a pod migrates. This default behavior also supports context-aware applications such as network discovery tools that need to know about IP addresses changes to operate correctly.

Persistent network interfaces assign the internal address equal to a static value that does not change due to pod migration. This is useful for applications that require the IP address that they see to remain unchanged in order for their open network connections to continue to function correctly. Although such applications are in the minority, one such application that is widely-used is FTP, which explicitly checks the source and destination IP addresses used for consistency for its protocol interaction. The static value that is used can be assigned in almost any manner. By default, the internal address seen by a connection is equal to the external IP address in use when the connection was first opened and persists for the lifetime of the given connection. Alternatively, the internal address can be statically assigned to a predefined value that is constant across all connections. This would be useful for a multi-homed web server, which is configured with multiple IP addresses and performs a different action depending on the IP address from which a request is made. By having predefined internal addresses, the web server can be started in a pod with the same configuration regardless of the host machine on which the pod is located. This can simplify web server administration for web hosting providers running web servers on a cluster of homogeneous machines.

To maintain an application's open connections persistently even after the application migrates, Zap needs to ensure that the network state perceived by the transport protocol for a given connection remains the same before and after migration, even though the pod's external IP address needs to change when its location changes. To address this problem, Zap incorporates a novel virtual networking mechanism that transparently supports persistent open end-to-end connections among migrating pods. The mechanism is based on the Virtual Network Address Translation (VNAT) architecture presented in [39], tailored specifically for Zap.

The idea behind VNAT is surprisingly simple. A virtual address, rather than a physical address, is introduced to identify a pod for its end-to-end connections. We use the term "address" loosely to refer to both an IP address

and port number, both of which are virtualized by VNAT. To send data over the connection, the virtual address is then translated into an appropriate physical address after packets leave the transport protocol and before they are injected into the network. Conversely, to receive data over the connection, a physical address is translated back into the corresponding virtual address before packets are returned to the transport protocol layer. Using these connection virtualization and translation mechanisms, a pod can migrate from place to place without changing the network connection state visible to the transport protocol. Note that Zap network virtualization differs somewhat from other resource virtualization as Zap must virtualize resources not just below the application layer, but below the transport layer as well.

Connection virtualization works by intercepting system calls for connection setup requests from the application to the transport protocol and replacing relevant physical addresses with virtual addresses. The result is that the transport protocol stack on both the client and the server will perceive a virtual connection with virtual addresses rather than a physical connection with the actual physical addresses of the machines. This virtual connection identification will stay unchanged for the life of the connection no matter where the pod containing the client or the server moves.

Connection translation works by intercepting packets below the transport protocol layer and translating the virtual addresses in the packet headers to physical addresses. A packet with a virtual address header is not routable on the physical network. Using connection translation, Zap translates a packet with a virtual address header sent by the client transport protocol into a packet with a physical address header so it can reach the intended server. Although this is similar to Network Address Translation (NAT), a key difference is that connection translation works entirely within the endpoint without introducing any connection state inside the network. Therefore connection translation does not suffer from many pitfalls experienced by traditional NAT [15, 16, 37].

To migrate a pod with open network connections, Zap checkpoints network state pertinent to its open network connections including standard operating system states such as socket structures and transport protocol states such as TCP PCB, as well as connection virtualization and translation states created by Zap. During the restart, the standard operating system states and transport protocol states are simply restored to their original values and the restarted pod can trivially locate the server location using the existing connection state. However, connection virtualization and translation states need to be updated on both endpoints to reflect the mapping between the (constant) internal address and the new external address of the migrated pod. When a connection endpoint resumes after migration, Zap notifies the endpoint on the other side of the connection that the migrated endpoint is at a new physical address. Both endpoints of the connection then update their virtualization state so that their virtual address pairs map to the new physical address pair. The protocol used to update the endpoints is detailed in [39]. Note that the virtual connection perceived by the transport protocol stays intact across the migration and the transport layer is completely unaware of the change of the underlying physical address of the client. So with the addition cost of translating a virtual connection to and from a physical connection, Zap will seamlessly migrate a transport end-to-end connection regardless of where the client moves.

Zap selects the virtual address for a connection to be the same as the current physical address for the connection, which corresponds to the current external IP address of the pod. This choice of virtual address provides two key advantages. First, it eliminates the need for the client and server to exchange their virtual addresses at connection setup time. Second, it eliminates connection translation overhead for connections that are not migrated. As a result, no translation overhead will ever be imposed on a connection so long as the pod does not move. After a pod migrates, only existing connections that have migrated along with the pod will incur connection translation overhead. New connections from the migrated pod will not incur such overhead since the virtual address used for the new connections will always be based on the current external address of the pod.

Zap distinguishes between the virtual address seen by the transport layer and the internal address seen by applications running within a pod. Since Zap intercepts system calls that return network addresses to applications, Zap can return any network address that it chooses. This makes it simple to support both transient and persistent network interfaces as discussed previously. For transient network interfaces, Zap returns the underlying external IP address to applications. For default persistent network interfaces, Zap propagates the virtualization up to applications and returns the virtual address seen by the transport layer when a connection was first established; for persistent network interfaces with statically assigned internal addresses, Zap simply remembers the static assignment and returns that value as the internal address.

In addition to providing connection persistency between applications running in pods, Zap can also provide support for preserving connections between pods and traditional processes running outside of pods assuming that such connections are made through a proxy. The network virtualization and migration mechanism of Zap can be installed separately from pods on a proxy. Zap network virtualization preserves a connection between a pod

and such a proxy the same way it preserves a connection between two pods. When the pod migrates, the connections between the pod and the proxy continue to be "spliced" with the connections between the proxy and the legacy applications behind the proxy. Since the proxy doesn't detect the movement of the pod due to Zap network virtualization, there is no need to "switch" the connections between the pod and the proxy. As a result, the complete proxied connections between the pod and traditional processes without pods are migrated without any modifications to applications or system environments without pods.

4.6 Pod Administration and Usage

Zap enables any user to create a pod, either explicitly or by incorporating pod creation into system utilities such as `login` to encapsulate a user's computing session within a pod. Once a pod is created, access to the pod is controlled by an access control list (ACL). Only the host system administrator on the system where the pod is currently executing or a user on the ACL are allowed to manipulate the pod, including suspending and resuming a pod for migration purposes. The five primary commands provided by Zap for users and system administrators to create and manipulate pods are:

create_pod enables a user to create a new pod with various options that can be specified in a pod configuration file. These options include the network configuration and file system configuration for the pod, what applications if any should be launched once the pod is created, and access control permissions for the pod. `create_pod` assigns a numerical identifier to the pod and creates a corresponding entry for the pod in a list maintained by Zap that contains a list of pods currently running on the host machine and associated information about each pod. The pod identifier may change when a pod migrates to another machine. Pod creation does not nest, so that creating a pod from within an existing pod will create a new pod on the host machine. Some examples of how `create_pod` can be used include: with `login` to create a user session in a pod, with `/etc/init.d` for automatically starting up services like a web server in a pod, or with `inetd` to spawn incoming connection handlers into a new pod, such as creating a separate pod for each `telnet` session.

kill_pod takes a pod identifier and terminates the respective pod, killing all processes in the pod, freeing all associated resources, and removing the pod itself from the system.

addproc_pod takes an executable and a pod identifier and creates a new process running the given executable in context of the respective pod. Note that `addproc_pod` only creates a new process in the pod; it will not move an already existing process into the pod to avoid creating naming conflicts. To add a process to a pod, Zap must first create a new process. In Unix systems, process creation is done using `fork`, which creates a child process that is a copy of the parent by allocating a kernel process structure and populating it with information from the parent. Zap creates a process in a similar fashion but does not use the same kind of information from the parent in the new process to ensure that there are no dependencies on the parent process or the host system or pod in which the parent process resides. Instead, Zap takes several steps to avoid creating any such dependencies in the new process, including setting its parent process to `init`, making it the process group leader, and relinquishing the control terminal.

Because adding a process to a pod is done in a system call, all of the necessary steps can be done in the kernel before the process is made runnable. In particular, the executable that `addproc_pod` specifies to run is overlaid on the new process before returning from the system call, not as a separate `exec` system call. Furthermore, the executable that `addproc_pod` executes is specified in the context of the virtual file system of the pod into which the process is being added, not the file system of the environment from which the `addproc_pod` command was issued. Finally, `addproc_pod` also creates an entry for the new process in the pod's process list. Although special care must be taken in adding a process to a pod, process creation inside of a pod is simply done in the normal manner with a created child process inheriting the attributes of its parent. Once a process has been added to a pod, all its operations occur inside of the pod, and all of its children will also be created inside of the pod.

suspend_pod takes a pod identifier and filename, stops the pod and all of its processes, checkpoints the state of the pod to the respective file.

resume_pod takes a filename for a file that contains a checkpointed pod and restarts the pod and its processes starting at the point at which the pod was checkpointed. `resume_pod` assigns a new numerical identifier to the pod.

To simplify administration, Zap provides a view of the host machine outside of the context of pods that shows everything running on the given machine in the same manner used in existing operating system environments. An administrator can access the host machine directly to obtain this administrative view. Processes within pods are viewable from this administrative view, but they cannot be accessed from this view using traditional IPC mechanisms to avoid creating host dependencies. We note that pods introduce interesting security considerations, but due to space constraints, these security issues are not discussed here.

5 Implementation

The Zap architecture was designed to provide transparent process migration while minimizing changes to the operating system by leveraging the loadable kernel module interface available in many commodity operating systems. We have implemented a Zap prototype as a Linux kernel module. Our implementation builds on previous work of one of the authors on a Linux kernel module called CRAK [42] that provided a restricted process migration mechanism but did not support the general pod abstraction. Our Zap implementation can be dynamically loaded on a running Linux system to provide Zap functionality without modifying, recompiling, or reinstalling the Linux kernel. We highlight some key mechanisms that were used for implementing Zap virtualization and migration in a kernel module.

Zap virtualization was largely implemented by providing a mechanism for intercepting system calls to translate between pod and operating system namespaces. Intercepting a system call within a Linux module is fairly simple. The module need only replace the appropriate system call handler pointer in the system call table by a pointer to the new system call handler. In order to invoke the previous system call handler, the new handler need only call the old function pointer. This results in a small amount of additional overhead due to the extra procedure call. For simple system calls like `getpid`, the only extra cost beyond the procedure call is a hash table lookup for translation. For more complex system calls like `fork`, Zap also needs to allocate pod-specific structures for keeping track of necessary process state for processes running in a pod. Note that when pods are running, system calls are also intercepted for processes running outside of any pod to check to make sure that those processes are not attempting to communicate directly with processes in a pod using local mechanisms.

Our Zap virtualization implementation also uses NFS for file system virtualization and Linux *netfilter* for network virtualization. Zap creates the virtual file system of a pod by mounting various NFS mount points from a file server for pods to a staging directory on the local machine. Zap uses the netfilter system in the Linux 2.4 series kernel, which is a packet filtering and mangling system [34]. Netfilter instruments the IP protocol stack at well-defined points during the traversal of the stack by a packet. It provide hooks that invoke user-registered functions to process the packet at these well-define points. Zap uses these hooks for source and destination address translation on both incoming and outgoing network traffic.

Zap migration was implemented by providing mechanisms to read and write kernel state to checkpoint and restart pods. Since kernel modules run in kernel mode, a Zap kernel module will have the necessary privileges to read and write kernel state that must be saved and restored for checkpointing and restarting pods. However, locating the process-related data structures that needed to be saved and restored was more difficult because Linux does not export all of the kernel structures to kernel modules. As it turns out, when the Linux kernel is built and distributed, there is a file called `System.map` which contains a list of all symbols, both exported and not, and their locations in the kernel memory space. Therefore, Zap simply queries this file to identify the addresses of structures which contained data to be checkpointed for the processes, as well as functions which were integral to the restoration of the processes.

One interesting decision we encountered while developing Zap's migration mechanism was whether the kernel structures should be saved in their native format or the individual elements from the structures saved. We decided to save the individual elements to enable Zap to migrate processes across minor version changes of kernels, even if the layout of their structures change slightly. In order to achieve cross-version migration, Zap must be able to translate the process and kernel state from the format provided by the source host to the format required by the target host. Not only must Zap be able to ensure compatibility between the versions, but it must also be capable of populating the target host with whatever values that were not provided by the source host. Although we have successfully used Zap to migrate pods across Linux kernels with minor version differences, a more complete examination of this issue is the subject of future work.

6 Experimental Results

We present some experimental results using our Linux Zap prototype on various applications. We conducted our experiments on a trio of IBM Netfinity 4500R machines, each with a 933 MHz Intel Pentium-III CPU, 512 MB RAM, 9.1 GB SCSI HD, and 100 Mbps Ethernet connected to a 3Com Superstack II 3900 switch. One of the machines was used as an NFS server from which directories were mounted to construct the virtual file system for a pod. All of the machines were installed with the RedHat Linux 7.1 distribution and the Linux 2.4.10 kernel.

Since VMMs have been proposed for migration, we also performed our experiments with VMware Workstation 3.2 for Linux with the same RedHat distribution and Linux kernel running in a VM. This provides a conservative comparison of our unoptimized prototype against a tuned commercial product. Unless otherwise indicated, the VM was configured with raw disk mode, bridged networking, and the recommended 384 MB of memory. While VMware did not allow us to configure the VM with the same memory size as the host RAM, we ensured that memory size was not a limitation for our experiments.

Section 6.1 describes some simple experiments to measure the cost of Zap virtualization compared with a vanilla Linux system and VMware. Section 6.2 describes examples of how pods can be used to provide mobile thin-client computing sessions and web servers, and measures the cost of migrating these sessions using Zap versus VMware.

6.1 Zap Virtualization

To measure the cost of Zap virtualization, we used a range of micro benchmarks and application benchmarks to measure both individual system call performance as well as real application performance. The nine benchmarks we used are described in Table 2. All of the benchmarks measure the time it takes to run the respective benchmark. volano is VolanoMark 2.1.2, an industry standard Java chat server benchmark configured in accordance with the rules of the Volano Report [4], but reports the average time per message transferred rather than the message transfer rate.

Name	Description	Linux
getpid	`getpid` run in a loop 10000 times, measure average iteration time	352 ns
shmget +shmctl	IPC shared memory segment holding an integer is created and removed	42 µs
semget +semctl	IPC semaphore variable is created and removed	19 µs
fork+ exit	process forks and waits for child which calls exit immediately	111 µs
fork+ execve	process forks and waits for child to run C program that prints “hello world” then exits	1811 µs
fork+ /bin/sh	process forks and waits for child to run `/bin/sh` to run C program that prints “hello world” then exits	7963 µs
volano	VolanoMark 2.1.2 using Java 2 Runtime Environment SE 1.4.1	219 µs/ mesg
make	Linux kernel compile with up to ten processes active at one time	440 s
apache	Netscape browser downloads Java-script-controlled sequence of 54 web pages from Apache 2.0.35 web server	16 s

Table 2: Application benchmarks

To obtain accurate measurements, we rebooted the systems between measurements and directly used the TSC register [3] available on Pentium CPUs to record timestamps at the significant measurement events. The average cost of each timestamp was 32 ns. The files for these benchmarks were stored on the NFS server for all of our experiments to provide a consistent comparison. We measured the performance of these benchmarks on four different Linux 2.4.10 system configurations:

- Linux - benchmarks are run on a vanilla Linux system to measure baseline system performance.
- VMware - benchmarks are run on vanilla Linux (guest OS) inside a VM on a vanilla Linux system to measure performance using a VM.
- With Pod - benchmarks are run on a Linux system with Zap installed and a pod created to measure performance on the host outside of a pod.
- Inside Pod - benchmarks are run in a pod on a Linux system with Zap installed to measure performance inside of a pod.

Table 2 shows the results of running the nine benchmarks on the vanilla Linux system. Figure 1 shows the results of running the benchmarks on the other three system configurations, the results normalized to the vanilla Linux system with the value one representing the normalized vanilla Linux results. Since all benchmarks measure the time to run the benchmark, a small number is better for all benchmark results in Figure 1.

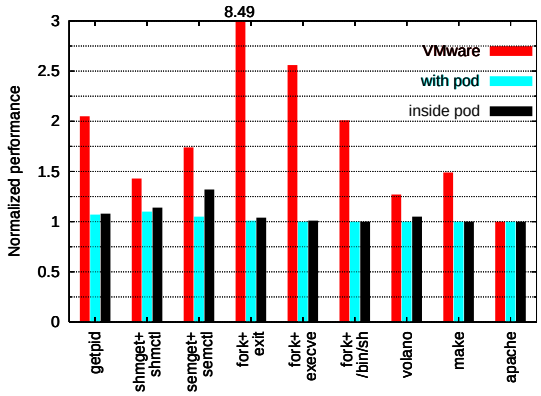

Figure 1: Virtualization cost

The results in Figure 1 show that VMware performs the worst on all of the benchmarks. The simple getpid benchmark takes more than twice as long using VMware compared to vanilla Linux. fork+exit gives the worst performance on VMware, running more than eight times slower than vanilla Linux. VMware does better on the benchmarks that are not dominated by system calls, but still runs 100% slower on fork+/bin/sh, 20% slower on volano, 50% slower on make compared to vanilla Linux. The only benchmark on which VMware does not perform worse is apache, where a web browser on another machine downloads and displays a sequence of 54 web pages from an Apache server running inside VMware. For apache, all of the system configurations delivered essentially the same performance.

Figure 1 shows that Zap virtualization overhead is quite small, especially compared with using a virtual machine monitor like VMware. When running inside a

pod, Zap overhead for the simple system call getpid benchmark was only 8% compared to vanilla Linux, reflecting the fact that Zap virtualization for these kinds of system calls only requires an extra procedure call and a hash table lookup. The most expensive benchmark for Zap was semget+semctl, which took 32% longer than vanilla Linux. The cost reflects the fact that our untuned Zap prototype needs to allocate memory and do a number of namespace translations. semget+semctl and shmget+shmctl both take 6 µs longer with Zap than vanilla Linux, but this accounts for a higher percentage of time for semget+semctl because it takes roughly half as much time overall. volano overhead with Zap is 5% more than vanilla Linux due to the overhead of using `clone` to generate high thread counts. There was no additional overhead for running make inside a pod compared to running on vanilla Linux.

Figure 1 also shows that the cost of running pods is also quite small for processes that are running on a host system outside of pods. With pods present, Zap must intercept system calls made by processes outside pods to ensure that they do not attempt to manipulate processes inside pods. The overhead of this pod protection was less than 10% compared to vanilla Linux for the benchmarks dominated by system call cost, namely getpid, shmget+shmctl, semget+semctl, and fork+exit. More importantly, there was no difference in overall performance between running with pods and vanilla Linux for any of the other larger application benchmarks that we tested.

6.2 Zap Migration

To illustrate how Zap can be used for different applications and measure the cost of migration using Zap, we used two different applications listed in Table 3, a VNC thin-client computing user session and a web server. The VNC [32] thin-client computing user session provides an example of how Zap can be used to provide mobility of a user's computing session. Table 3 shows that eleven legacy and network X applications were run as part of the VNC session for our experiments. The apache session provides an example of how Zap can be used to provide web server mobility. We encapsulated each of the two sessions in a pod and measured the cost of suspending each pod and resuming it on another host machine, both in terms of the time to checkpoint and restart the pods and in terms of the amount of state that needs to be saved for migration. For comparison, we also measured the cost of suspending and resuming these two application sessions on the same machine using VMware. We did not actually migrate the sessions using VMware because it does not support migration of networked applications.

The results of migrating the application sessions using Zap and suspending and resuming them using VMware are shown in Figure 2. For these experiments, the sessions were checkpointed to and restarted from an image on the local disk. For VMware, we ran experiments with two VM configurations, one with 128 MB of memory and the other with 384 MB of memory. In all cases, the resulting checkpoint image was always a little more than the memory size given to the VM, regardless of what applications were running. The time to suspend the sessions grew disproportionately with the memory size given to the VM, taking about 2 seconds per session for a 128 MB VM, but more than 25 seconds per session for a 384 MB VM.

Name	Applications
VNC	Xvnc 3.3.3 - VNC virtual X server twm - window manager Netscape communicator 4.76 - web browser telnet - telnet client inside `xterm` window, connected to another machine on the same LAN xterm+bash - shell running in `xterm` window xview - image viewer w/ 13 KB GIF loaded xcalc - X-based calculator xclock - analog clock xman - X-based man page browser xelvis - `vi` text editor w/ 870 byte text file loaded xpdf - PDF viewer w/ 293 KB 14-page file PDF loaded
apache	Apache 2.0.35 - web server, default number of worker processes used

Table 3: Application sessions

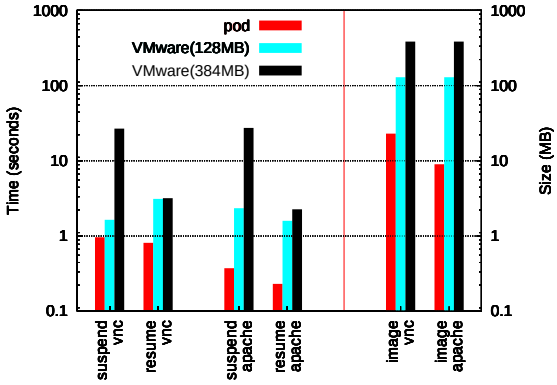

Figure 2: Migration cost

Our results show that Zap saves much less state and is much faster than VMware in suspending and resuming a running application session. Figure 2 shows that checkpoint and restart times for each pod were less than a second. Checkpointing the VNC pod took 963 ms and resulted in 23 MB of image data, whereas checkpointing the apache pod took 373 ms and resulted in 9 MB of image data. Memory contents accounted for over 99% of the checkpoint image file sizes, but only grow with the applications actually used, as opposed to VMware in

which the sizes grow with the RAM allocated to the VM. If the medium over which the checkpoint images are transferred for migration were slow or somehow limited in capacity, the checkpointed pod images could be compressed further using gzip, resulting in a 4.6 MB VNC pod image and a 0.9 MB apache pod image. Restarting the pods was slightly faster. Restarting the VNC pod took 811 ms and restarting the apache pod took 231 ms. The restart times are faster than checkpoint times in part because parts of the image files are mapped directly to memory during restart and are loaded by the operating system as they fault, whereas the checkpoint process needs to save all state.

We further analyzed the amount of time that Zap spent checkpointing and restarting each application within the VNC pod. The time to checkpoint an application in a pod was generally bound by the resulting image size. The application with the largest checkpoint time was Netscape, which had 10 MB of memory contents which needed to be saved. Most other application checkpoint times varied with the amount of memory pages which needed to be saved, suggesting that the checkpoint times are primarily I/O bound. However, the telnet application restart time accounted for a disproportionate amount of the time to restart the VNC pod, especially given its modest contribution of 363 KB to the 23 MB pod image size. The reason for this is because restarting this network application required a round-trip message to the remote end of the connection to inform it of the new location and set up translation rules. Nevertheless, the overall cost of restarting telnet in its `xterm` window was still modest at only about 140 ms. This time is not accounted for in the VMware measurements since the VMware could not migrate the VNC session and maintain the telnet connection.

Our results for migrating pods running realistic applications show that pod migration costs were modest overall, with subsecond checkpoint and restart times for the VNC and apache pods. More importantly, our results demonstrate the ability of Zap to migrate legacy applications without modification, including graphical X applications and networked applications such as telnet.

7 Conclusions and Future Work

Zap is the first system that we are aware of that provides transparent migration of legacy and networked applications across machines running independent operating systems without requiring any changes to the operating systems. Zap achieves this behavior by leveraging loadable kernel module technology and introducing a thin virtualization layer that decouples applications from host dependencies in the operating system. Zap introduces and supports a pod abstraction, which encapsulates groups of processes in a virtualized environment that can be migrated as a unit. We have implemented Zap as a Linux kernel module to demonstrate the viability of our approach. Our experimental results on real applications using our Linux Zap prototype show that Zap can provide general-purpose process migration functionality with low overhead. We hope that Zap will provide a useful tool and building block for exploring the benefits and applications of migratable computing environments.

Zap raises a number of interesting follow-up research areas. First, Zap raises many interesting questions, both in terms of security mechanisms that should be implemented within Zap itself as well as security mechanisms and policies that should be considered for systems hosting pods. Current security schemes do not generally take into account process migration; as such, more work should be done to properly understand the issues and how they may be addressed. In addition, process migration is most beneficial when used under the appropriate circumstances. This raises the question of when to migrate a pod, and which pod to migrate. Finally, support for additional devices remains an open question because each device has its own requirements. As such, additional study into how common devices should be handled is warranted.

8 Acknowledgments

Eric Brewer, our paper shepherd, Brian Schmidt, and Erez Zadok provided many helpful comments on earlier drafts of this paper. Shaya Potter contributed to Zap design discussions and parts of the Linux Zap implementation. This work was supported in part by NSF grants EIA-0071954 and ANI-0117738, an NSF CAREER Award, and an IBM SUR Award.

9 References

[1] *http://www.vmware.com*, VMware, Inc.

[2] *NFS: Network File System Protocol Specification*, RFC1094, Sun Microsystems, Inc., March 1989.

[3] *Using the RDTSC Instruction for Performance Monitoring*, Pentium II Processor Application Notes, Intel Corporation, 1997.

[4] *The Volano Report*, Volano LLC, December 2001. http://www.volano.com/report

[5] K. Amiri, D. Petrou, G. Ganger, and G. Gibson, *Dynamic Function Placement in Active Storage Clusters*, Technical Report CMU-CS-99-140, School of Computer Science, Carnegie Mellon University, June 1999.

[6] Y. Artsy, Y. Chang, and R. Finkel, *Interprocess Communication in Charlotte*, IEEE Software:22-28, January 1987.

[7] A. Barak and R. Wheeler, *MOSIX: An Integrated Multiprocessor UNIX*, Proceedings of the USENIX Winter 1989 Technical Conference, pp. 101-112, San Diego, CA, February 1989.

[8] P. Bhagwat, C. Perkins, and S. K. Tripathi, *Network Layer Mobility: an Architecture and Survey*, IEEE Personal Communication, **3**(3):54-64, June 1996.

[9] T. Boyd and P. Dasgupta, *Process Migration: A Generalized Approach Using a Virtualized Operating System*, Pro-

ceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS 2002), Vienna, Austria, July 2002.

[10] J. Casas, D. L. Clark, R. Conuru, S. W. Otto, R. M. Prouty, and J. Walpole, *MPVM: A Migration Transparent Version of PVM*, Computing Systems, **8**(2):171-216, 1995.

[11] D. Cheriton, *The V Distributed System*, Communications of the ACM, **31**(3):314-333, March 1988.

[12] F. Douglis and J. Ousterhout, *Transparent Process Migration: Design Alternatives and the Sprite Implementation*, Software - Practice and Experience, **21**(8):757-785, August 1991.

[13] I. Foster and C. Kesselman, *Globus: A Metacomputing Infrastructure Toolkit*, Proceedings of the Workshop on Environments and Tools for Parallel Scientific Computing, Lyon, France, August 1996.

[14] A. Grimshaw and W. Wulf, *The Legion Vision of a Worldwide Virtual Computer*, Communications of the ACM, **40**(1):39-45, January 1997.

[15] T. Hain, *Architectural Implications of NAT*, RFC2993, IETF, November 2000.

[16] M. Holdrege and P. Srisuresh, *Protocol Complications with the IP Network Address Translator*, RFC3027, IETF, January 2001.

[17] D. B. Johnson and C. Perkins, *Mobility Support in IPv6*, draft-ietf-mobileip-ipv6-16.txt, IETF, March 2002.

[18] A. D. Joseph, J. A. Tauber, and M. F. Kaashoek, *Mobile Computing with the Rover Toolkit*, IEEE Transactions on Computers, **46**(3):337-352, March 1997.

[19] E. Jul, *Migration of Light-weight Processes in Emerald*, IEEE Technical Committee on Operating Systems Newsletter, **3**(1):20-23, 1989.

[20] J. Jung, E. Sit, H. Balakrishnan, and R. Morris, *DNS Performance and the Effectiveness of Caching*, Proceedings of ACM SIGCOMM Internet Measurement Workshop, pp. 153-167, San Francisco, CA, November 2001.

[21] M. Kozuch and M. Satyanarayanan, *Internet Suspend/Resume*, Fourth IEEE Workshop on Mobile Computing Systems and Applications, Calicoon, NY, June 2002.

[22] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny, *Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System*, Technical Report #1346, University of Wisconsin Madison Computer Sciences, April 1997.

[23] D. A. Maltz and P. Bhagwat, *MSOCKS: An Architecture for Transport Layer Mobility*, Proceedings of the IEEE INFOCOM'98, pp. 1037-1045, San Francisco, CA, 1998.

[24] D. Milojicic, F. Douglis, and R. Wheeler, *Mobility: Processes, Computers, and Agents*, Addison Wesley Longman, February 1999.

[25] S. J. Mullender, G. v. Rossum, A. S. Tanenbaum, R. v. Renesse, and H. v. Staveren, *Amoeba – A Distributed Operating System for the 1990s*, IEEE Computer, **23**(5):44-53, May 1990.

[26] C. Perkins, *IP Mobility Support for IPv4, revised*, draft-ietf-mobileip-rfc2002-bis-08.txt, Internet Draft, September 2001.

[27] R. Pike, D. Presotto, K. Thompson, and H. Trickey, *Plan 9 from Bell Labs*, Proceedings of the Summer 1990 UKUUG Conference, pp. 1-9, London, July 1990.

[28] J. S. Plank, M. Beck, G. Kingsley, and K. Li, *Libckpt: Transparent Checkpointing under Unix*, Proceedings of Usenix Winter 1995 Technical Conference, pp. 213-223, New Orleans, LA, January 1995.

[29] J. Pruyne and M. Livny, *Managing Checkpoints for Parallel Programs*, 2nd Workshop on Job Scheduling Strategies for Parallel Processing (In Conjunction with IPPS '96), Honolulu, Hawaii, April 1996.

[30] X. Qu, J. X. Yu, and R. P. Brent, *A Mobile TCP Socket*, International Conference on Software Engineering (SE '97), San Francisco, CA, November 1997.

[31] R. Rashid and G. Robertson, *Accent: a Communication Oriented Network Operating System Kernel*, Proceedings of the 8th Symposium on Operating System Principles, pp. 64–75, December 1984.

[32] T. Richardson, Q. Stafford-Fraser, K. Wood, and A. Hopper, *Virtual Network Computing*, IEEE Internet Computing, **2**(1):33-38, January 1998.

[33] M. Rozier, V. Abrossimov, F. Armand, M. Gien, M. Guillemont, F. Hermann, and C. Kaiser, *Chorus (Overview of the Chorus Distributed Operating System)*, Proceedings of the USENIX Workshop on Micro-Kernels and other Kernel Architectures, Seattle, WA, April 1992.

[34] R. Russell, *Linux 2.4 Packet Filtering HOWTO*, Linux Netfilter Core Team, November 2001. http://netfilter.samba.org/

[35] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum, *Optimizing the Migration of Virtual Computers*, Proceedings of the 5th Operating Systems Design and Implementation, Boston, MA, December 2002.

[36] B. K. Schmidt, *Supporting Ubiquitous Computing with Stateless Consoles and Computation Caches*, Ph.D Thesis, Computer Science Department, Stanford University, August 2000.

[37] D. Senie, *Network Address Translator (NAT)-Friendly Application Design Guidelines*, RFC3235, IETF, January 2002.

[38] A. C. Snoeren and H. Balakrishnan, *An End-to-End Approach to Host Mobility*, Proceedings of 6th International Conference on Mobile Computing and Networking (MobiCom'00), Boston, MA, August 2000.

[39] G. Su and J. Nieh, *Mobile Communication with Virtual Network Address Translation*, Technical Report CUCS-003-02, Department of Computer Science, Columbia University, February 2002.

[40] P. Vixie, S. Thomson, Y. Rekhter, and J. Bound, *Dynamic Updates in the Domain Name System (DNS UPDATE)*, RFC2136, IETF, April 1997.

[41] Y. Zhang and S. Dao, *A "Persistent Connection" Model for Mobile and Distributed Systems*, 4th International Conference on Computer Communications and Networks (ICCCN), Las Vegas, NV, September 1995.

[42] H. Zhong and J. Nieh, *CRAK: Linux Checkpoint/Restart As a Kernel Module*, Technical Report CUCS-014-01, Department of Computer Science, Columbia University, November 2001.